**Introduction**

Building a state-of-art verification environment for a design is essentially building a large complex software system. Many constructs and techniques from computer programming are supported by the verification language for convenience in development. However, some commonly used ones are not. For instance, "alias" is not defined in the SystemVerilog Standard. Despite the reasons regarding not having "alias" in the language, in some cases, aliasing can be very handy for designing an efficient testbench. It helps to reduce the maintenance effort and prevents human mistakes while building or using the verification environment.

**Problems in the Design**

Here are the two typical problems when designing and using the UVM environment:

a) Extra actions for packing and unpacking the operands. For example, the driver needs to decode the opcode then pack the fields from the command transaction into the operand before sending it into the interface. Vice versa for the Monitor.

b) Data synchronization between variables is not seamless. For example, in the command transaction class, the same information is stored in two independent sets of variables. Updating either of them does not imply updating the other one.

**Potential Alternative Solution**

**A. Standard-Defined "alias"**

SystemVerilog 3.1 has introduced "alias" into the standard. The alias statement models a bidirectional short-circuit connection used in a module or an interface construct, by which the designer can assign different "virtual" nets sharing the same physical net. Here is a code example of the "alias" usage in an interface:

```verilog
interface Command_Transaction_A
  virtual function void pack_operand();
  virtual function void unpack_operand();
  bit [31:0] operand;
endinterface

typedef union Joined
  signed field_A0, field_A1:8;
  s unsigned field_A2:8;
endunion

class Command_Transaction_A extends Base_Transaction;
  typedef Joined bit [31:0] operand;
endclass

class Command_Transaction_B extends Base_Transaction;
  typedef Joined bit [31:0] operand;
endclass
```

As mentioned previously, "alias" is not allowed in a class construct in SystemVerilog. The pseudo codes will fail to be compiled. Therefore, we will take the journey of seeking for the solutions to workaround this and hopefully achieve the desired goal in the testbench design.

**B. let Construct**

SystemVerilog from the IEEE 1800-2012 standard introduces the "let" construct, which defines a template expression, customized by its ports. Besides customizing the text macros, it can be used to provide shortcuts for identifiers. It can be used in UVM by declaring it in a SystemVerilog package. However, the let identifier cannot be used in the left-hand-side (LHS) of an assignment as shown in the example below. So the user will not be able to update the operand through assigning values directly to the fields in the command transactions. Here is an example:

```verilog
virtual function void pack_operand();
  op = 1;
  bit [31:0] operand;
  return operand;
endfunction
```

This polymorphism technique eliminates the casting actions, which saves some code and solves problems of bit-packing in the testbench development. However, it still relies on the functions to synchronize the data between variables. Therefore, the users and the testbench designers need to carefully decide when and where to call these functions to avoid data corruption.

**The User-Defined Aliasing Method**

After exhaust the potential solutions from the SystemVerilog Standard, it is concluded that none of them can fully resolve the problem in the testbench construction for this design. Therefore, let's come back to the aliasing method and see if this can be implemented in a class construct for the verification environment. In a computer system, aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program. Hence, implementing a user-defined alias in a SystemVerilog class construct needs to retrieve the memory address of the variable first. But SystemVerilog does not define pointer type and has no system function to retrieve the memory address of a variable either. In this case, to leverage the fine establishment of the pointer referencing in C++, the DPI-C plugin can be used to work with SystemVerilog for this problem. Here is an example of the DPI-C functions and the usage in SystemVerilog environment:

```c
#include <uvm_macros.h>
#include <uvm_test.h>

typedef union Joined
  signed field_A0, field_A1:8;
  s unsigned field_A2:8;
endunion

class Command_Transaction_A
  typedef Joined bit [31:0] operand;
endclass

class Command_Transaction_B
  typedef Joined bit [31:0] operand;
endclass

int get_pAddress(int* variable) {
  int offset = 0;
  int pAddress = 0;
  pAddress = (int)variable;
  return pAddress;
}

int main() {
  int* variable = ...;
  int pAddress = get_pAddress(variable);
  ...}
```

**Summary**

Though the DPI-C approach has some solvable limitations, it empowers the verification environment with the "alias" method. Different variables can be linked dynamically to reduce the effort in testbench development and maintenance. With an insignificant run-time overhead, the verification environment can become more scalable, less design error-prone, and more user-friendly.